LA-UR-21-31286
Approved for public release; distribution is unlimited.

Title:	Partial program correctness
Author(s):	Herring, Stuart Davis
Intended for:	WG21 teleconferences Web
Issued:	2021-11-12

P1494R2: Partial program correctness

Audience: EWG; CWG; LEWG; WG14
S. Davis Herring <herring@lanl.gov>
Los Alamos National Laboratory
November 10, 2021

History

r2:

- Discussed the preexisting problem of input dependence
- Discussed interactions with other kinds of observable behavior
- Discussed lack of debugger support
- Fixed (preexisting) “first” in wording based on SG1 review

r1:

- Made introduction explicitly independent of contracts
- Fixed conditionals in example

Problem

Undefined behavior enables and extends many important optimizations (*e.g.*, simplifying signed integer arithmetic and dead-code elimination). The “time travel” aspect of such optimizations (explicitly authorized by [intro.abstract]/5) is surprising to many programmers in that it can sometimes eliminate tests meant to detect the invalid operation in question. In particular, consider

```
#include<cstdio>
#include<cstdlib>

static void bad(const char *msg) {
  std::fputs(msg, stderr);
#ifdef DIE
  std::abort();
#endif
}

void inc(int *p) {
  if(!p) bad("Null!\n");
  ++*p;
}
```

Without `-DDIE`, a conforming implementation can elide the test in `inc` entirely: `std::fputs` always returns, so any call `inc(nullptr)` is guaranteed to have undefined behavior and need not call `bad`. (Note that current implementations do not do so in this case.)

This issue came up again recently in the discussion of contracts:

```
void f(int *p) [[expects: p]] [[expects: *p<5]];
```

Discomfort with the idea that (with continuation mode “on”) the first *contract-attribute-specifier* might be elided because of the second was one of the motivations for the many late proposals to change (and eventually remove) contracts. Many wondered about the possibility of making a contract violation handler “opaque to optimization”, so that the first precondition must be checked on the supposition that the handler might not return (but rather throw or terminate).

The capability of establishing such a “checkpoint”, where subsequent program behavior, *even if undefined*, does not affect the preceding behavior, would be useful in general for purposes of stability and debugging.

There is already an analogous issue concerning program input: clearly a program can have undefined behavior for some inputs and not others. [intro.abstract]/3 and /5 acknowledge this by referring to “a given input” and “that input”, but such a monolithic approach neglects the paradoxical possibility of input correlated with, say, unspecified quantities:

```
int x[1];

int main() {
  std::uintptr_t a=reinterpret_cast<std::uintptr_t>(x)%3+1,b;
  std::cerr << "The car is behind door number " << a
```

```
              << ".  Door to open: ";
  std::cin >> b;
  return x[a-b];
}
```

This program might suggest 2 (if the cast yields, say, 0x1000), but one could absurdly argue that responding with 2 leads to undefined behavior because almost all of the unspecified possibilities for the "address" of `x` (*e.g.*, 0x2000) lead to undefined behavior for that input. An OOTA interpretation is also available: "`a` is really 3, so the input 2 leads to undefined behavior, whose effect is to print 2 instead of `a`". We should reject this on the grounds of causality, since we require that undefined behavior respect input which can quite reasonably depend on *prior* output.

The standard does not have the necessary notion of *prior*, and this paper does not address this situation, but its checkpoints may be used to strengthen the guarantee of /6.3, that prompts are delivered before waiting for input, to include all observable behavior. This usage does exclude the OOTA interpretation: the output delivered to the host environment must accurately represent `a` if it is separated by a checkpoint from any potential undefined behavior.

Previous work

I suggested a trick involving a `volatile` variable, based on the idea that a volatile read is *observable behavior* ([intro.abstract]/6) that must be preserved by optimization.

```
inline void log(const char *msg)
{std::fputs(msg, stderr);}    // always returns

bool on_fire() {
  static volatile bool fire;  // always false
  return fire;
}

void f(int *p) {
  if (p == nullptr) log("bad thing 1");
  if (on_fire()) std::abort();
  if (*p >= 5) log("bad thing 2");
}
```

The idea is that the compiler cannot assume that `on_fire()` returns `false`, and so the check for `p` being null cannot be eliminated. However, the compiler can observe that, if `p` is null, the behavior will be undefined *unless* `on_fire()` returns `true`, and so it can elide that check (though not the volatile read) and call `abort()`. This therefore seems to convey a certain capability of *observing* the upcoming undefined behavior without actually experiencing it.

Unfortunately, conforming implementations are not constrained to follow this analysis. It is logically necessary that the implementation perform the observable volatile read unless it can somehow obtain its result otherwise. However, after reading the value `false` (as of course it will be in practice) the implementation may take any action whatsoever, even "undoing" the call to `log`. For example, it would be permissible to perform the implicit flush for `stderr` only just before the call to `std::abort` (which never happens). One might hope for the implementation to allow for the possibility that `log` affects some hardware state that affects the volatile read, but it might not as such a scheme would require support from the operating system.

General solution

We can instead introduce a special library function

```
namespace std {
  // in <cstdlib>
  void observable() noexcept;
}
```

that divides the program's execution into *epochs*, each of which has its own observable behavior. If any epoch completes without undefined behavior occurring, the implementation is required to exhibit the epoch's observable behavior. Ending an epoch is nonetheless distinct from ending the program: for example, there is no automatic flushing of `<cstdio>` streams.

Undefined behavior in one epoch may obscure the observable behavior of a previous epoch (for example, by re-opening an output file), but external mechanisms such as pipes to a logging process can be used to guarantee receipt of an epoch's output. With multiple threads, it is not the epochs themselves that are meaningful but their boundaries (or *checkpoints*); normal thread synchronization is required for the observable behavior of one thread to be included in an checkpoint defined by another.

As a practical matter, a compiler can implement `std::observable` efficiently as an intrinsic that counts as a possible termination, which the optimizer thus cannot remove. After optimization (including any link-time optimization), the code generator can then produce zero machine instructions for it.

Note that `std::observable` does not itself constitute observable behavior, and it does not forgive infinite empty loops ([intro.progress]/1). There

is no explicit connection to volatile access, but the ordinary happens-before rules apply (as much as possible given the vacuous [intro.abstract]/6.1). Finally, there is no guarantee that, for instance, local variables have been spilled to registers at each checkpoint: `std::observable` prevents certain program reorderings, but it is not a general aid for comprehensibility when using a debugger. (In general, it is difficult if not impossible to specify semantics that allow optimization and yet behave correctly when presented with input from a user equipped with a debugger.)

Limited assumptions

A call to `std::observable` prevents the propagation of assumptions based on the potential for undefined behavior after it into code before it. The following functions offer the same opportunities for dead-code elimination:

```
void a(int &r, int *p) {
  if (!p) std::fprintf(stderr, "count: %d\n", ++r);
  if (!p) std::abort();  // henceforth, p is known to be non-null
  if (!p) std::fprintf(stderr, "p is null\n");
}
```

```
void b(int &r, int *p) {
  if (!p) std::fprintf(stderr, "count: %d\n", ++r);
  std::observable();
  if (!p) std::fprintf(stderr, "p is null\n");
  *p += r;               // p may be assumed non-nu
}
```

In both cases, the “p is null” output can be elided: in `a`, because execution would not continue past the `std::abort`; in `b`, because of the following dereference of `p`. In both cases, the `count` output must appear if `p` is null: in `a`, because the program thereafter has the defined behavior of aborting; in `b`, because the epoch ends before undefined behavior occurs.

The function `b`, however, offers the additional optimization of not checking for null pointers at run time. It is very useful to support such optimizations without compromising diagnostics.

Usage

The obvious place to use `std::observable` is after any sort of I/O that always returns, especially in any code run when an error is detected (and so imminent undefined behavior is likely). In a contracts context, the violation handler is one such routine; since `std::observable()` has no side effects, it would also be permissible to include it in specific contract conditions to guarantee that previous contracts are checked (even if the violation handler always returns):

```
void f(int *p) [[expects: p]] [[expects: (std::observable(), *p<5)]];
```

Wording

Relative to N4901.

Add a paragraph before [intro.abstract]/5:

- An *observable checkpoint* is a call to `std::observable` ([support.start.term]) or program termination.

Change [intro.abstract]/5 as follows:

- A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this document places ~~no~~ requirements on the implementation executing that program with that input ~~(not even with regard to operations preceding the first~~ for only those operations *O* for which for every undefined operation~~)~~ *U* there is an observable checkpoint *C* such that *O* happens before *C* and *C* happens before *U*.

  [*Note:* The undefined behavior that arises from a data race ([intro.races]) happens on all participating threads. — *end note*]

Change [intro.abstract]/6.2 as follows:

- ~~At program termination~~ At each observable checkpoint, all data whose delivery to the host environment to be written ~~into files~~ to any file happens before that checkpoint shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced. [*Note:* Not all host environments provide access to file contents before program termination. — *end note*]

[*Drafting note:* The phrase “delivery to … any file” refers to C11 7.21.5.2/2. — *end note*]

Add to [cstdlib.syn]:

- `[[noreturn]] void quick_exit(int status) noexcept;`

  `void observable() noexcept;`

Add paragraphs to the end of [support.start.term]:

- `void observable() noexcept;`

- *Effects:* Establishes an observable checkpoint ([intro.abstract]). No other effects.